Online Design Bug Detection

Paper by Kypros Constantinides, Onur Mutlu, Todd Austin
published in MICRO 2008

David Kleymann

Seminar in Computer Architecture 2020

November 26, 2020
Executive Summary

- **Problem:**
  - Increasing complexity of modern CPUs makes Design Bugs in commercial products more common
  - They are hard to fix/avoid in software and usually unfixable in hardware

- **Goal:**
  - develop hardware solutions that enables detecting when a Design Bug triggered
  - has to be flexible to detect new bugs as they are discovered
Executive Summary

- **Contributions:**
  - in-depth **study of design bugs** of a quasi-commercial CPU at a low level
  - novel mechanism to **monitor** internal CPU signals and deciding whether a Design Bug can be triggered
  - Makes hardware ”updatable” with bug patches like software

- **Evaluation:**
  - To cover 80% of all bugs found in the study:
    - low power overhead *(3.5%)*
    - moderate area overhead *(10%)*
  - when combined with Hardware Fault Detection, some hardware can be shared and total overhead reduces
Presentation Outline

Paper Summary
  Problem
  Study of Design Bugs
  Proposed solution
  Major results
  Summary

Analysis
  Strengths
  Weaknesses

Discussion
Paper Summary

Problem
Study of Design Bugs
Proposed solution
Major results
Summary

Analysis

Strengths
Weaknesses

Discussion
Problem

- Modern **CPUs are highly complex**, especially CISC architectures
- A lot of effort goes into verifying designs before production, can take more than 50% of the release cycle
Problem

- Modern **CPUs are highly complex**, especially CISC architectures
- A lot of effort goes into verifying designs before production, can take more than 50% of the release cycle
- Design **bugs still appear** in widespread commercial CPUs
Problem

- Modern **CPUs are highly complex**, especially CISC architectures
- A lot of effort goes into verifying designs before production, can take more than 50% of the release cycle
- Design **bugs still appear** in widespread commercial CPUs
- Bugs in CPUs make it less usable: Correct software on buggy hardware can produce wrong result
Modern CPUs are highly complex, especially CISC architectures
A lot of effort goes into verifying designs before production, can take more than 50% of the release cycle
Design bugs still appear in widespread commercial CPUs
Bugs in CPUs make it less usable: Correct software on buggy hardware can produce wrong result
In commercial CPUs, bugs also lead to bad press and expensive recalls
Many bugs in the past, were usually handled by trying to avoid in software or disabling CPU components
Examples

- Intel FDIV bug: Intel Pentium can return wrong floating-point division results
  - Resulted in 500M $ recall
- Intel F00F bug - certain instruction with the right arguments locks up entire system
- AMDs have bugs too - a lot of consecutive pops and rets can cause some AMD Opterons to incorrectly update stack pointer
Examples

- Intel FDSV bug: Intel Pentium can return wrong floating-point division results
  - Resulted in 500M $ recall
- Intel F00F bug - certain instruction with the right arguments locks up entire system
Examples

- Intel FDIV bug: Intel Pentium can return wrong floating-point division results
  - Resulted in 500M $ recall
- Intel F00F bug - certain instruction with the right arguments locks up entire system
- AMDs have bugs too - a lot of consecutive pops and rets can cause some AMD Opterons to **incorrectly update stack pointer**
Goal

• First step to avoiding Design Bugs is to detect when a bug is triggered.
• In online operation.
• Optimally, we want to detect all Design Bugs.
• Not all bugs will be discovered at the manufacture date of the CPU.
   → We want to be able to add information about design bugs subsequently.
Goal

• First step to avoiding Design Bugs is to detect when a bug is triggered
Goal

- First step to avoiding Design Bugs is to detect when a bug is triggered
- In Online operation
Goal

- First step to avoiding Design Bugs is to detect when a bug is triggered
- In Online operation
- Optimally, we want to detect all Design Bugs
Goal

- First step to avoiding Design Bugs is to **detect** when a bug is triggered
- In Online operation
- Optimally, we want to detect all Design Bugs
- Not all bugs will be discovered at the manufacture date of the CPU
Goal

- First step to avoiding Design Bugs is to **detect** when a bug is triggered
- In **Online** operation
- Optimally, we want to detect all Design Bugs
- Not all bugs will be discovered at the manufacture date of the CPU
- → We want to be able to **add** information about design bugs **subsequently**
Paper Summary

Problem
Study of Design Bugs
Proposed solution
Major results
Summary

Analysis
Strengths
Weaknesses

Discussion
How do design bugs look like?

Paper analyzes OpenSPARC T1 RTL-level implementation and comes up with three categories:

- **Algorithmic design bugs**
  - Major deviation of implemented algorithm from specification
  - Involves a lot of buggy logic, detecting and fixing is usually hard

- **Logic design bugs**
  - buggy logic block(s) used somewhere (e.g. wrong type of gate, wrong combination of inputs)
  - Fixing and detection is easier, since erroneous hardware localized to a few gates

- **Timing design bugs**
  - Signal latched at the wrong time
  - Often fixed by adding/removing a buffer flip-flop
How do design bugs look like?

Paper analyzes OpenSPARC T1 RTL-level implementation and comes up with three categories:

- **Algorithmic design bugs**
  - **major deviation** of implemented algorithm from specification
  - Involves a lot of buggy logic, detecting and fixing is usually hard

- **Logic design bugs**
  - buggy logic block(s) used somewhere (e.g. wrong type of gate, wrong combination of inputs)
  - Fixing and detection is easier, since erroneous hardware localized to a few gates

- **Timing design bugs**
  - Signal latched at the wrong time
  - Often fixed by adding/removing a buffer flip-flop
How do design bugs look like?

Paper analyzes OpenSPARC T1 RTL-level implementation and comes up with three categories:

- **Algorithmic design bugs**
  - **major deviation** of implemented algorithm from specification
  - Involves a lot of buggy logic, detecting and fixing is usually hard

- **Logic design bugs**
  - buggy logic block(s) used somewhere (e.g. wrong type of gate, wrong combination of inputs)
  - Fixing and detection is easier, since erroneous hardware localized to a few gates
How do design bugs look like?

Paper analyzes OpenSPARC T1 RTL-level implementation and comes up with three categories:

- **Algorithmic design bugs**
  - major deviation of implemented algorithm from specification
  - Involves a lot of buggy logic, detecting and fixing is usually hard

- **Logic design bugs**
  - buggy logic block(s) used somewhere (e.g. wrong type of gate, wrong combination of inputs)
  - Fixing and detection is easier, since erroneous hardware localized to a few gates

- **Timing design bugs**
  - Signal latched at the wrong time
  - Often fixed by adding/removing a buffer flip-flop
What about manufacturing defects?

This paper is concerned with Logic design bugs. What does this paper not try to detect?
What about manufacturing defects?

This paper is concerned with Logic design bugs. What does this paper not try to detect?

- Algorithmic or Timing Design bugs
- Hardware faults caused by manufacturing process or deterioration of hardware
- Bugs and interference vulnerabilities of physical nature (things like Rowhammer) which are hard to detect
- Needs to be detectable by monitoring internal CPU signals
What about manufacturing defects?

This paper is concerned with **Logic design** bugs. What does this paper **not** try to detect?

- Algorithmic or Timing Design bugs
- **Hardware faults** caused by manufacturing process or deterioration of hardware
What about manufacturing defects?

This paper is concerned with **Logic design** bugs. What does this paper **not** try to detect?

- Algorithmic or Timing Design bugs
- **Hardware faults** caused by manufacturing process or deterioration of hardware
- Bugs and interference vulnerabilities of **physical nature** (things like Rowhammer) which are hard to detect
What about manufacturing defects?

This paper is concerned with **Logic design** bugs. What does this paper **not** try to detect?

- Algorithmic or Timing Design bugs
- **Hardware faults** caused by manufacturing process or deterioration of hardware
- Bugs and interference vulnerabilities of **physical nature** (things like Rowhammer) which are hard to detect
- Needs to be detectable by **monitoring internal CPU signals**
Logic design bugs
Logic design bugs

- Most common type of design bug in OpenSPARC T1
Logic design bugs

- **Most common type** of design bug in OpenSPARC T1
- 99% of all design bugs in two CPU sections: LSU (left) and TLU (right)

![Pie Chart 1](Logic 59 %)

- Algorithmic 35 %
- Timing 6 %

![Pie Chart 2](Algorithmic 47 %)

- Algorithmic 47 %
- Timing 4 %
Logic design bugs

- **Most common type** of design bug in OpenSPARC T1
- 99% of all design bugs in two CPU sections: LSU (left) and TLU (right)

![Pie charts showing logic, algorithmic, and timing bugs]

- **Hard to discover** in verification phase, if bug only occurs in very specific states
Logic design bugs

- **Most common type** of design bug in OpenSPARC T1
- 99% of all design bugs in two CPU sections: LSU (left) and TLU (right)

Algorithmic: 35%

- Logic: 59%
- Timing: 6%

Algorithmic: 47%

- Logic: 49%
- Timing: 4%

- **Hard to discover** in verification phase, if bug only occurs in very specific states
- Once discovered, **easy to detect** by monitoring source signals
Logic design bugs

- **Most common type** of design bug in OpenSPARC T1
- 99% of all design bugs in two CPU sections: LSU (left) and TLU (right)

![Pie chart showing Algorithmic 35% and Logic 59% for LSU, and Algorithmic 47% and Logic 49% for TLU.]

- **Hard to discover** in verification phase, if bug only occurs in very specific states
- Once discovered, **easy to detect** by monitoring source signals
- Algorithmic and Timing bugs could be easier to find in design verification
Example logic design bug

Buggy code:

```plaintext
assign buggy_signal = foo & ~(rst | hw_int | sr_int);
```

Correct code:

```plaintext
assign buggy_signal = foo & ~(rst | sr_int);
```
What do we learn from this example?

- Semantically, bug occurs on specific combinations of **First-Level Signals**
What do we learn from this example?

- Semantically, bug occurs on specific combinations of **First-Level Signals**
- These **might not exist** in finished CPU
What do we learn from this example?

- Semantically, bug occurs on specific combinations of **First-Level Signals**
- These **might not exist** in finished CPU
- But because we are at RTL-level it suffices to monitor the **Source-Level signals** corresponding to the First-Level Signals

![Diagram showing module inputs, source-level signals, first-level signals, and combinational logic with a red box highlighting buggy logic.](image-url)
Scale of monitored signals

- On OpenSPARC T1 there are usually less than 64 Source-Level Signals per bug
Scale of monitored signals

- On OpenSPARC T1 there are usually less than 64 Source-Level Signals per bug
- On average 9 of those are not shared with any other bug
Scale of monitored signals

- On OpenSPARC T1 there are usually less than 64 Source-Level Signals per bug
- On average 9 of those are not shared with any other bug
- In total, **1118 signals** to be monitored for detection of all 162 (documented) logic design bugs
On OpenSPARC T1 there are usually less than 64 Source-Level Signals per bug

On average 9 of those are not shared with any other bug

In total, **1118 signals** to be monitored for detection of all 162 (documented) logic design bugs

This is bad news!
Scale of monitored signals

- On OpenSPARC T1 there are usually less than 64 Source-Level Signals per bug
- On average 9 of those are not shared with any other bug
- In total, **1118 signals** to be monitored for detection of all 162 (documented) logic design bugs
- This is bad news!
- None of the logic design bugs in T1 had source signals from data or bus registers, only control signal registers
Paper Summary

Problem
Study of Design Bugs

Proposed solution
Major results
Summary

Analysis
Strengths
Weaknesses

Discussion
Basic idea - Signatures

- **Triggering conditions** for a bug represented by *Bug Signature*
Basic idea - Signatures

- **Triggering conditions** for a bug represented by *Bug Signature*
- *Bug Signatures* express what values **Source-Level signals** need to have for the bug to occur (0, 1, X - don’t care)
Basic idea - Signatures

- **Triggering conditions** for a bug represented by *Bug Signature*

- *Bug Signatures* express what values *Source-Level signals* need to have for the bug to occur (0,1, X - don’t care)

- Bug Signatures for all bugs combined into **single System Bug Signature**

![Bug Signature Collection Diagram](image-url)
Basic idea - Segments

- **Bug Detection Segments** monitor signals (flip-flops) they are responsible for.
Basic idea - Segments

- Bug Detection Segments monitor signals (flip-flops) they are responsible for.
- Each of those outputs whether all its signals match System Bug Signature.

![Diagram showing Bug Signature Collection, System Bug Signature, and Segment Checking Tree].
Basic idea - Segments

- **Bug Detection Segments** monitor signals (flip-flops) they are responsible for
- each of those outputs whether all its signals match System Bug Signature
- **Bug Detection Segment** match results are combined using Segment Match Detection Tables into a Segment Checking Tree to generate Global Bug Detection Signal

- Only System Bug Signature and Segment Match Detection Tables need to be field-programmable
- Firmware updates can then initialize these
Basic idea - Segments

- Only System Bug Signature and Segment Match Detection Tables need to be field-programmable

![Diagram of Bug Detection Segments and Segment Match Detection Tables](image)
Basic idea - Segments

- Only System Bug Signature and Segment Match Detection Tables need to be **field-programmable**
- Firmware updates can then initialize these
Why do we need a Segment Checking Tree?

- We can’t just ”OR” the bug signatures together, that would lead to too many false positives
Why do we need a Segment Checking Tree?

- We can’t just ”OR” the bug signatures together, that would lead to too many false positives
- summarize (”OR”) all bug signatures of a single design bug to form an Intermediate Bug Signature
Why do we need a Segment Checking Tree?

- We can’t just ”OR” the bug signatures together, that would lead to too many false positives
- summarize (”OR”) all bug signatures of a single design bug to form an Intermediate Bug Signature
- Merge Intermediate Signatures to System Bug Signature in a special way (see example)
Why do we need a Segment Checking Tree?

- We can’t just ”OR” the bug signatures together, that would lead to too many false positives
- summarize (”OR”) all bug signatures of a single design bug to form an Intermediate Bug Signature
- Merge Intermediate Signatures to System Bug Signature in a special way (see example)
- Bug Detection Segments that do need to report a signature match for a certain design bug to occur are selected in Segment Match Detection Tables
Why do we need a Segment Checking Tree?

- We can’t just “OR” the bug signatures together, that would lead to too many false positives
- Summarize (“OR”) all bug signatures of a single design bug to form an Intermediate Bug Signature
- Merge Intermediate Signatures to System Bug Signature in a special way (see example)
- Bug Detection Segments that do need to report a signature match for a certain design bug to occur are selected in Segment Match Detection Tables
  - This is essentially demultiplexing the System Bug Signature
Why do we need a Segment Checking Tree?

- We can’t just "OR" the bug signatures together, that would lead to too many false positives
- summarize ("OR") all bug signatures of a single design bug to form an Intermediate Bug Signature
- Merge Intermediate Signatures to System Bug Signature in a special way (see example)
- Bug Detection Segments that do need to report a signature match for a certain design bug to occur are selected in Segment Match Detection Tables
  - This is essentially demultiplexing the System Bug Signature
- Tree structure is needed to reduce number of false positives, while reducing space used on storing Bug Signatures
Signature merging example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Signature merging example

\[
\begin{array}{c|c|c|c|c}
X & 1 & 0 & 0 \\
X & 1 & 0 & 1 \\
\hline
X & 1 & 0 & X \\
\end{array}
\]
Signature merging example

<table>
<thead>
<tr>
<th>X</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

This segment will be ignored in a node of the checking tree.
Signature merging example

\[
\begin{array}{c|c|c|c}
X & 1 & 0 & 0 \\
\hline
X & 1 & 0 & 1 \\
\hline
X & 1 & 0 & X \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
X & X & X & X & X \\
\hline
X & X & X & X & X \\
\hline
X & X & X & X & X \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
X & X & X & X & X \\
\end{array}
\]
Signature merging example

\[
\begin{array}{cccc}
X & 1 & 0 & 0 \\
X & 1 & 0 & 1 \\
\hline
X & 1 & 0 & X
\end{array}
\]

\[
\begin{array}{cccc}
X & X & X & X \\
X & X & X & X \\
\hline
X & X & X & X
\end{array}
\]

this segment will be ignored in a node of the checking tree
Signature merging example

\[
\begin{array}{c|cccc}
\times & 1 & 0 & 0 & \\
\times & 1 & 0 & 1 & \\
\hline
\times & 1 & 0 & x & \\
\end{array}
\]

this segment will be ignored in a node of the checking tree
Signature merging example

\[
\begin{array}{c|cc}
X & 1 & 0 & 0 \\
- & - & - & 0 \\
X & 1 & 0 & 1 \\
- & - & - & 1 \\
X & 1 & 0 & X \\
- & - & - & X \\
\end{array}
\]

\[
\begin{array}{c|cccc}
X & X & X & X & X \\
- & - & - & - & X \\
X & X & X & X & X \\
- & - & - & - & X \\
X & X & X & X & X \\
- & - & - & - & X \\
\end{array}
\]

\[
\begin{array}{c|cccc}
X & 1 & 0 & X \\
- & - & - & - & X \\
X & X & X & X & X \\
- & - & - & - & X \\
\end{array}
\]

this segment will be ignored in a node of the checking tree
Signature merging example

this segment will be ignored in a node of the checking tree
Hardware implementation

• Idea: integrate Bug Detection Segments into flip-flops
  • keeps routes short, output compared directly at flip-flops
  • System Bug Signature translates into two signals per flip-flop: 0/1 and care/don't care (X)
  • Bug Detection logic in Flip-flops outputs 0 for a signature match, and 1 for a mismatch
  • All flip-flops in one Bug Detection Segment have their local bug detection signals chained together with OR-gates → Only if all flip-flop's values match signature, Bug Detection Segment sends match signal up the Segment Checking Tree
Hardware implementation

- Idea: integrate Bug Detection Segments into flip-flops

- Keeps routes short, output compared directly at flip-flops

- System Bug Signature translates into two signals per flip-flop: 0/1 and care/don’t care (X)

- Bug Detection logic in Flip-flops outputs 0 for a signature match, and 1 for a mismatch

- All flip-flops in one Bug Detection Segment have their local bug detection signals chained together with OR-gates → Only if all flip-flop’s values match, Bug Detection Segment sends match signal up the Segment Checking Tree
Hardware implementation

- Idea: integrate Bug Detection Segments into flip-flops
- keeps routes short, output compared directly at flip-flops
Hardware implementation

- Idea: integrate Bug Detection Segments into flip-flops
- keeps routes short, output **compared directly** at flip-flops
- System Bug Signature translates into **two signals per flip-flop**: 0/1 and care/don’t care(X)
Hardware implementation

- Idea: integrate Bug Detection Segments into flip-flops
- keeps routes short, output **compared directly** at flip-flops
- System Bug Signature translates into **two signals per flip-flop**: 0/1 and care/don’t care(X)
- Bug Detection logic in Flip-flops outputs 0 for a signature match, and 1 for a mismatch
Hardware implementation

- Idea: integrate Bug Detection Segments into flip-flops
- keeps routes short, output **compared directly** at flip-flops
- System Bug Signature translates into **two signals per flip-flop**: 0/1 and care/don’t care(X)
- Bug Detection logic in Flip-flops outputs 0 for a signature match, and 1 for a mismatch
- All flip-flops in one Bug Detection Segment have their local bug detection signals chained together with OR-gates
  → Only if **all** flip-flop’s values **match** signature, Bug Detection Segment **sends match** signal up the Segment Checking Tree
Reuse of Scan chain logic

- Modern CPUs use scan flip-flops, an augmented flip-flop type that can be used for hardware testing
Reuse of Scan chain logic

- Modern CPUs use scan flip-flops, an augmented flip-flop type that can be used for hardware testing
- Allows all flip-flops of the processor to be connected in a Scan chain (like a large shift register) and to be tested using ATPG
Reuse of Scan chain logic

- Modern CPUs use scan flip-flops, an augmented flip-flop type that can be used for hardware testing
- Allows all flip-flops of the processor to be connected in a Scan chain (like a large shift register) and to be tested using ATPG
- Used once after fabrication, after that scan logic is inactive
Reuse of Scan chain logic

- Modern CPUs use scan flip-flops, an augmented flip-flop type that can be used for hardware testing
- Allows all flip-flops of the processor to be connected in a Scan chain (like a large shift register) and to be tested using ATPG
- Used once after fabrication, after that scan logic is inactive
- Use **scan logic to load one bit** of System Bug Signature to flip-flops, use additional logic to store the other bit
Extensions and additional aspects

For actually fixing/avoiding bug after detection, existing checkpointing-based recovery solutions such as ReVive or SafetyNet can be used.

Can be neatly combined with similar online hardware fault detection (“Access/Control Extension”) to share even more hardware.

Paper proposes mechanism to tweak false positive rate.
Extensions and additional aspects

- For actually **fixing/avoiding** bug after detection, existing checkpointing-based **recovery solutions** such as ReVive or SafetyNet can be used
Extensions and additional aspects

- For actually **fixing/avoiding** bug after detection, existing checkpointing-based **recovery solutions** such as ReVive or SafetyNet can be used.
- Can be neatly combined with similar online **hardware fault detection** ("Access/Control Extension") to share even more hardware.
Extensions and additional aspects

- For actually **fixing/avoiding** bug after detection, existing checkpointing-based **recovery solutions** such as ReVive or SafetyNet can be used.
- Can be neatly combined with similar online **hardware fault detection** ("Access/Control Extension") to share even more hardware.
- Paper proposes mechanism to tweak false positive rate.
Thoughts

• Can we modify the system to detect Timing Bugs?
Thoughts

- Can we modify the system to detect Timing Bugs?
  - We could make Segment Match Detection nodes (of the tree) only propagate their results on each clock cycle
Thoughts

• Can we modify the system to detect Timing Bugs?
  • We could make Segment Match Detection nodes (of the tree) only propagate their results on each clock cycle
  • Then chain nodes together in a way that we match a temporal pattern
Thoughts

• Can we modify the system to detect Timing Bugs?
  • We could make Segment Match Detection nodes (of the tree) only propagate their results on each clock cycle
  • Then chain nodes together in a way that we match a temporal pattern
  • But this would mean getting rid of the 'levels' of the Segment Checking tree
Thoughts

• Can we modify the system to detect Timing Bugs?
  • We could make Segment Match Detection nodes (of the tree) only propagate their results on each clock cycle
  • Then chain nodes together in a way that we match a temporal pattern
  • But this would mean getting rid of the 'levels' of the Segment Checking tree
  • Also detection of non-Timing Bugs would be delayed by a number of cycles (bad considering a bug could lock up the CPU in the meantime)
Paper Summary

- Problem
- Study of Design Bugs
- Proposed solution
- Major results
- Summary

Analysis

- Strengths
- Weaknesses

Discussion
Evaluation method

• Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
• Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
• Covers all control-signal FFs except for memories/caches
• Caches and most other parts of CPU evaluated using simulation tools
• Power consumption of some parts was taken from UltraSPARC T1 specs
Evaluation method

- Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
Evaluation method

- Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
- Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
Evaluation method

• Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
• Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
• Covers all control-signal FFs except for memories/caches
Evaluation method

- Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
- Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
- Covers all control-signal FFs except for memories/caches
- Power consumption of some parts was taken from UltraSPARC T1 specs
Evaluation method

- Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
- Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
- Covers all control-signal FFs except for memories/caches
- Caches and most other parts of CPU evaluated using simulation tools
Evaluation method

- Using RTL design of OpenSPARC T1 for evaluation of area and power overhead
- Augment design with implementation of bug detection flip-flops, segment checking tree with field programmable match detection tables
- Covers all control-signal FFs except for memories/caches
- Caches and most other parts of CPU evaluated using simulation tools
- Power consumption of some parts was taken from UltraSPARC T1 specs
Fixing design parameters

• To precisely estimate overhead of design, design parameters have to be fixed first.
• Paper chooses 8-bit Bug Detection Segments, 4-level tree structure.
Fixing design parameters

• To precisely estimate overhead of design, design parameters have to be fixed first
Fixing design parameters

- To precisely estimate overhead of design, \textit{design parameters} have to be fixed first
- Paper chooses \textbf{8-bit} Bug Detection Segments, \textbf{4-level tree} structure
Fixing design parameters

Table 3. Fraction of data and control signals in the OpenSPARC T1

<table>
<thead>
<tr>
<th>Segment Match Detection Entries Per Node</th>
<th>Design Bugs Covered (%)</th>
<th>Percentage of Critical Design Bugs (64%) [20]</th>
<th>Area Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

Figure 16. Area overhead versus design bug coverage

- How many Segment Match Detection Table entries?
Fixing design parameters

How many Segment Match Detection Table entries?

Paper chooses 16 entries (*80% bug coverage*), arguing that not all design bugs are critical.
Fixing design parameters

Table 3. Fraction of data and control signals in the OpenSPARC T1

Figure 16. Area overhead versus design bug coverage

- How many Segment Match Detection Table entries?
- Paper chooses 16 entries (**80% bug coverage**), arguing that not all design bugs are critical
- In a quoted comparison of other CPUs, **only about 64%** of all bugs were **critical**
Fixing design parameters

How many Segment Match Detection Table entries?

Paper chooses 16 entries (80% bug coverage), arguing that not all design bugs are critical

In a quoted comparison of other CPUs, only about 64% of all bugs were critical

Non-critical = errors in performance measurement, error reporting, debugging etc.
Evaluation results - Area

→ With 16 entries, we get silicon area overhead of 10%
Evaluation results - Area

→ With 16 entries, we get silicon area overhead of 10%
  • 17% area overhead for full bug coverage
Evaluation results - Power

• Baseline Power consumption estimated at 56.3 W (about 12% less than commercial UltraSPARC T1)
• Design with Online Bug detection - 58.3 W, 3.5% increase
• Amortized overhead when we add online hardware defect detection (ACE):
  • 15.15% area and 6.8% power
Evaluation results - Power

- Baseline Power consumption estimated at 56.3 W (about 12% less than commercial UltraSPARC T1)
Evaluation results - Power

- Baseline Power consumption estimated at 56.3 W (about 12% less than commercial UltraSPARC T1)
- Design with Online Bug detection - 58.3 W, **3.5% increase**
Evaluation results - Power

• Baseline Power consumption estimated at 56.3 W (about 12% less than commercial UltraSPARC T1)
• Design with Online Bug detection - 58.3 W, **3.5% increase**
• **Amortized overhead** when we add online **hardware defect detection** (ACE):
  • 15.15% area and 6.8% power
Evaluation results - Power

- ACE Framework: 1.3%
- Segment Checking Tree: 1.5%
- Augmented Flip-Flops: 0.6%
- Cores & L1 Cache
- L2 Cache
- Leakage
- Crossbar, MC, Misc
- I/O Pads
- Wires & Repeaters
- FPU
Paper Summary

Problem
Study of Design Bugs
Proposed solution
Major results
Summary

Analysis

Strengths
Weaknesses

Discussion
Executive Summary

**Problem:**
- **Increasing complexity** of modern CPUs makes **Design Bugs** in commercial products **more common**
- They are hard to fix/avoid in software and usually unfixable in hardware

**Goal:**
- develop hardware solutions that enables **detecting** when a **Design Bug triggered**
- has to be **flexible** to detect new bugs as they are discovered
Executive Summary

• Contributions:
  • in-depth **study of design bugs** of a quasi-commercial CPU at a low level
  • novel mechanism to **monitor** internal CPU signals and deciding whether a Design Bug can be triggered
    • integrated into Flip-Flops, **reusing hardware** used in CPUs today, **field programmable**
    • **Variable amount** of detectable bugs (trade-off w/ area overhead), covering **all signals** of importance
    • Extensible to also do Hardware Fault Detection
  • Makes hardware ”updatable” with bug patches like software
    • **Less pressure on verification**, can make development of new CPUs faster

• Evaluation:
  • To cover 80% of all bugs found in the study:
    • low power overhead (**3.5%**)
    • moderate area overhead (**10%**)
  • when combined with Hardware Fault Detection, some hardware can be shared and total overhead reduces
Paper Summary
  Problem
  Study of Design Bugs
  Proposed solution
  Major results
  Summary

Analysis
  Strengths
  Weaknesses

Discussion
Strengths

• Based on thorough low-level analysis of design bugs, not based on processor errata sheets
• Previous work often makes assumptions based on (high-level) processor errata sheets
• Flexible and comprehensive solution
• Almost all signals can be covered
• Bug Signatures are "updatable"
• Low power overhead and moderate area overhead due to clever reuse of existing scan-chain logic
• Overhead can amortize in combination with Hardware Fault Detection
• Paper goes into a lot of detail, but is still intelligible
Strengths

- Based on thorough **low-level analysis** of design bugs, not based on processor errata sheets
  - Previous work often makes assumptions based on (high-level) processor errata sheets
Strengths

- Based on thorough **low-level analysis** of design bugs, not based on processor errata sheets
  - Previous work often makes assumptions based on (high-level) processor errata sheets
- Flexible and comprehensive solution
  - Almost **all signals** can be covered
  - Bug Signatures are "**updatable**"
Strengths

• Based on thorough **low-level analysis** of design bugs, not based on processor errata sheets
  • Previous work often makes assumptions based on (high-level) processor errata sheets
• Flexible and comprehensive solution
  • Almost **all signals** can be covered
  • Bug Signatures are "updatable"

• **Low power overhead** and moderate area overhead due to clever **reuse** of existing **scan-chain logic**
Strengths

• Based on thorough **low-level analysis** of design bugs, not based on processor errata sheets
  • Previous work often makes assumptions based on (high-level) processor errata sheets

• Flexible and comprehensive solution
  • Almost **all signals** can be covered
  • Bug Signatures are "**updatable**"

• **Low power overhead** and moderate area overhead due to clever **reuse** of existing **scan-chain logic**

• Overhead can amortize in combination with Hardware Fault Detection
Strengths

• Based on thorough low-level analysis of design bugs, not based on processor errata sheets
  • Previous work often makes assumptions based on (high-level) processor errata sheets
• Flexible and comprehensive solution
  • Almost all signals can be covered
  • Bug Signatures are "updatable"
• Low power overhead and moderate area overhead due to clever reuse of existing scan-chain logic
• Overhead can amortize in combination with Hardware Fault Detection
• Paper goes into a lot of detail, but is still intelligible
Paper Summary

Problem
Study of Design Bugs
Proposed solution
Major results
Summary

Analysis

Strengths
Weaknesses

Discussion
Weaknesses

• Bug analysis tailored to one particular CPU design - conclusions might not hold for other CPU designs.

• OpenSPARC/UltraSPARC T1 is in-order superscalar CPU, most competitor CPUs at that time already used out-of-order execution.

• Large category of Algorithmic Design Bugs is ignored on the basis that they might be discovered in verification.

• In TLU+LSU, 45% of the bugs were not Logic Design Bugs!

• Algorithmic Design Bugs can have greater impact than Logic Design Bugs.
Weaknesses

- Bug analysis tailored to one particular CPU design - conclusions might not hold for other CPU
Weaknesses

• Bug analysis **tailored** to one **particular CPU** design - conclusions might not hold for other CPU
  - OpenSPARC/UltraSPARC T1 is in-order superscalar CPU, most competitor CPUs at that time already used out-of-order execution
Weaknesses

• Bug analysis **tailored** to one **particular CPU** design - conclusions might not hold for other CPU
  • OpenSPARC/UltraSPARC T1 is in-order superscalar CPU, most competitor CPUs at that time already used out-of-order execution

• Large category of Algorithmic Design Bugs is **ignored** on the basis that they might be discovered in verification
Weaknesses

- Bug analysis tailored to one particular CPU design - conclusions might not hold for other CPU
  - OpenSPARC/UltraSPARC T1 is in-order superscalar CPU, most competitor CPUs at that time already used out-of-order execution
- Large category of Algorithmic Design Bugs is ignored on the basis that they might be discovered in verification
  - In TLU+LSU 45% of the bugs were not Logic Design Bugs!
Weaknesses

• Bug analysis *tailored* to one *particular CPU* design - conclusions might not hold for other CPU
  • OpenSPARC/UltraSPARC T1 is in-order superscalar CPU, most competitor CPUs at that time already used out-of-order execution
• Large category of Algorithmic Design Bugs is *ignored* on the basis that they might be discovered in verification
  • In TLU+LSU 45% of the bugs were *not* Logic Design Bugs!
  • Algorithmic Design Bugs *can have greater impact* than Logic Design Bugs
Weaknesses

- Power and area overhead evaluation are estimations based on partial simulation with different tools.

Criticality of bugs in OpenSPARC T1 was not analyzed.
Weaknesses

• Power and area overhead evaluation are estimations based on partial simulation with different tools
  • Exact overhead can only be measured after place-and-route
Weaknesses

- Power and area overhead evaluation are estimations based on partial simulation with different tools.
  - Exact overhead can only be measured after place-and-route.
  - Detailed RTL model is missing implementation of recovery mechanism/bug avoidance.
Weaknesses

- Power and area overhead evaluation are estimations based on **partial simulation** with different tools
  - Exact overhead can only be measured **after place-and-route**
  - Detailed RTL model is **missing** implementation of **recovery mechanism/bug avoidance**
  - Full solution will have **higher overhead**
Weaknesses

- Power and area overhead evaluation are estimations based on **partial simulation** with different tools
  - Exact overhead can only be measured **after place-and-route**
  - Detailed RTL model is **missing** implementation of **recovery mechanism/bug avoidance**
  - Full solution will have **higher overhead**
- Estimated overhead based on assumption that 80% bug coverage is enough
  - Criticality of bugs in OpenSPARC T1 was not analyzed
Discussion

• Are design bugs still an issue?
• Think about the current trend and the future - will the number of bugs in new CPUs increase?
• Or will the CPU designers learn from their mistakes and produce less design bugs?
• Can't fix everything with microcode patches?
• Complex ISA instructions are sometimes implemented using architectural microcode instructions.
• These can nowadays be patched to avoid some bugs.
• Think about the bugs you have seen: Are logic bugs directly tied to specific instructions?
• What about modern CPU security vulnerabilities (e.g. Spectre)?
Discussion

- Are design bugs still an issue?
Discussion

• Are design bugs still an issue?
  • Think about the current trend and the future - will the number of bugs in new CPUs **increase**?
  • Or will the CPU designers learn from their mistakes and produce **less** design bugs?
Discussion

- Are design bugs still an issue?
  - Think about the current trend and the future - will the number of bugs in new CPUs increase?
  - Or will the CPU designers learn from their mistakes and produce less design bugs?
- Can’t fix everything with microcode patches?
Discussion

- Are design bugs still an issue?
  - Think about the current trend and the future - will the number of bugs in new CPUs increase?
  - Or will the CPU designers learn from their mistakes and produce less design bugs?
- Can’t fix everything with microcode patches?
  - Complex ISA instructions are sometimes implemented using architectural microcode instructions
  - These can nowadays be patched to avoid some bugs
• Are design bugs still an issue?
  • Think about the current trend and the future - will the number of bugs in new CPUs **increase**?
  • Or will the CPU designers learn from their mistakes and produce **less** design bugs?
• Can’t fix everything with **microcode** patches?
  • Complex ISA instructions are sometimes implemented using architectural microcode instructions
  • These can nowadays be patched to avoid some bugs
  • Think about the bugs you have seen: Are logic bugs **directly tied** to specific instructions?
Discussion

• Are design bugs still an issue?
  • Think about the current trend and the future - will the number of bugs in new CPUs increase?
  • Or will the CPU designers learn from their mistakes and produce less design bugs?

• Can’t fix everything with microcode patches?
  • Complex ISA instructions are sometimes implemented using architectural microcode instructions
  • These can nowadays be patched to avoid some bugs
  • Think about the bugs you have seen: Are logic bugs directly tied to specific instructions?
  • What about modern CPU security vulnerabilities (e.g. Spectre)?
Discussion

- Is it okay to concentrate on **Logic Design Bugs** and ignoring Algorithmic/Timing ones?
Discussion

- Is it okay to concentrate on Logic Design Bugs and ignoring Algorithmic/Timing ones?
  - Are the bugs in other categories really more likely to be found before the processor is sold?
Discussion

• Is it okay to concentrate on Logic Design Bugs and ignoring Algorithmic/Timing ones?
  • Are the bugs in other categories really more likely to be found before the processor is sold?
• How could one detect Algorithmic Bugs without too many False Positives?
Discussion

• Is it okay to concentrate on **Logic Design Bugs** and ignoring Algorithmic/Timing ones?
  • Are the bugs in other categories really more likely to be found before the processor is sold?

• How could one detect **Algorithmic** Bugs without too many False Positives?
  • Can you apply tactics from algorithm verification? **Invariants** for algorithms in hardware?
Discussion

- Is it okay to concentrate on **Logic Design Bugs** and ignoring Algorithmic/Timing ones?
  - Are the bugs in other categories really more likely to be found before the processor is sold?
- How could one detect **Algorithmic** Bugs without too many False Positives?
  - Can you apply tactics from algorithm verification? **Invariants** for algorithms in hardware?
  - Actually papers investigating this exist

---

*SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs*

Matthew Hicks  
University of Michigan  
mhicks@umich.edu

Cynthia Sturton  
University of North Carolina at Chapel Hill  
costurton@cs.unc.edu

Samuel T. King  
Twitter, Inc.  
sking@twitter.com

Jonathan M. Smith  
University of Pennsylvania  
jms@cis.upenn.edu
End of presentation
0: Signature Match 1: Mismatch

DETECT_IN (from previous bug detection FF)

Bug Detection Portion

LD_BUG_SIGN

SCAN_CLK

SCAN_IN (from previous scan latch)

CAPTURE

UPDATE

D

CLK

Q

DETECT_OUT (to next bug detection FF)

1D Latch C1

1D Latch C1

Scan Portion

C1

1D Latch C2

2D

System Portion

1D

C1 Latch C1

2D

C2

C1
0: Signature Match 1: Mismatch

 DETECT_IN  (from previous bug detection FF)  DETECT_OUT  (to next bug detection FF)

 LD BUG SIGN

 Bug Detection Portion

 1D Latch C1

 1D Latch C1

 SCAN CLOCK

 SCAN IN  (from previous scan latch)  SCAN OUT  (to next latch in scan chain)

 CAPTURE

 Scan Portion

 1D Latch C1

 2D

 C1 Latch 1D

 UPDATE

 D

 System Portion

 1D Latch C1

 2D C2

 Q

 CLK
0: Signature Match 1: Mismatch

- **DETECT_IN** (from previous bug detection FF)
- **LD_BUG_SIGN**
- **Bug Detection Portion**
  - 1D Latch C1
  - 1D Latch C1
- **DETECT_OUT** (to next bug detection FF)

- **SCAN_CLK**
- **SCAN_IN** (from previous scan latch)
- **CAPTURE**
- **Scan Portion**
  - C1 Latch 1D
  - C1 Latch 1D

- **UPDATE**
- **D**
- **CLK**
- **System Portion**
  - 1D Latch C1
  - 1D Latch C2

- **Q**
- **SCAN_OUT** (to next latch in scan chain)
Signature merging 2

\[
\begin{array}{cc}
X & 1 & 0 & X \\
X & 1 & 0 & X \\
\end{array}
\]
Signature merging 2

\[
\begin{array}{cccc}
X & 1 & 0 & X \\
X & 1 & 0 & X \\
X & 1 & 0 & X \\
\hline
X & 1 & 0 & X \\
\end{array}
\]
Signature merging 2

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>1</th>
<th>0</th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>**</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>1</th>
<th>X</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Signature merging 2

\[
\begin{array}{cccc}
X & 1 & 0 & X \\
X & 1 & 0 & X \\
X & 1 & 0 & X \\
\end{array}
\]

\[
\begin{array}{cccc}
X & 1 & X & 1 \\
X & 1 & X & 0 \\
\end{array}
\]
Signature merging 2

\[
\begin{array}{c}
X & 1 & 0 & X \\
X & 1 & 0 & X \\
X & 1 & 0 & X \\
\hline
X & 1 & 0 & X \\
\end{array}
\]
Signature merging 2

\[
\begin{array}{c|c|c|c|c}
X & 1 & 0 & X \\
X & 1 & 0 & X \\
X & 1 & 0 & X \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
X & 1 & X & 1 \\
X & 1 & X & 0 \\
X & 1 & X & X \\
\end{array}
\]